XRay: A Function Call Tracing System

Authors:
Alistair Veitch (aveitch@google.com)
Dean Berris (dberris@google.com)
Eric Anderson (echoalfa@google.com)
Nevin Heintze (nch@google.com)
Ning Wang (nwang@google.com)
Contact:
google-xray@googlegroups.com

Date: 2016-04-05

Table of Contents

- Function Call Tracing in Production
- Introducing XRay
  - How XRay Works
  - XRay Overheads
- Implementation Details
  - Compiler-inserted Instrumentation Points
  - Runtime Logging and Support Library
  - Conversion and Analysis Tools
- Current Work and Future Plans

Abstract

Debugging high throughput, low-latency C/C++ systems in production is hard. At Google we developed XRay, a function call tracing system that allows Google engineers to get accurate function call traces with negligible overhead when off and moderate overhead when on, suitable for services deployed in production. XRay enables efficient function call entry/exit logging with high accuracy timestamps, and can be dynamically enabled and disabled. This white paper describes the XRay tracing system and its implementation. It also describes future plans with open sourcing XRay and engaging open source communities.

Function Call Tracing in Production

At Google we run distributed systems handling massive amounts of concurrent requests that do everything from serving a webpage to storing bytes on disks. Understanding the performance of these complex systems is difficult given the abstractions involved, asynchronous programming with callbacks, and even interaction between threads synchronising amongst each other. It can be especially difficult to track down what affects performance in rare events, which can be rare because they occur at the latency tail[1], or because they are outlier requests to the service. Seeing what happens "at the tail" allows software engineers to diagnose the performance of systems in these rare events that have an impact on the overall performance of the system.

The question then becomes how do you find these "needle in a haystack" events while running in production? Time-based traces let developers track down these issues when standard approaches such as sampling, logging, and aggregate statistics fail. What kind of time-based trace data would be useful?

Ideally, we'd like to get non-sampled function call traces with high precision timestamps from production servers that tell us which threads are running what code and when. However, instrumenting all functions is usually too expensive for production use. To deploy in production, we require that:

- The cost is acceptable when tracing and barely measurable when not tracing.
- Instrumentation is automatic and directed towards functions that are important for understanding the binary's execution time.
- Tracing is efficient in both space and time -- only recording what is required and what matters.

In addition to the above, we also require:

- Tracing is configurable with thresholds for storage (how much memory to use) and accuracy (whether to log everything or only function calls taking at least some amount of time).
- Tracing does not require changes to the operating system nor super-user privileges.
- Tracing can be turned on and off dynamically without having to restart the server.

[1] For more on why optimising tail latency matters, refer to "The Tail at Scale", by Jeff Dean and Luiz Barroso, published to the Communications of the ACM, vol. 56 (2013), pp. 74-80.

Introducing XRay

XRay is a function call tracing system developed internally at Google for debugging performance issues in production servers. XRay is not specific to a class of applications and is applicable to any C/C++-based binary -- from storage servers handling multiple thousands of requests per second to debugging command-line tools and unit tests. XRay is a suite of tools integrated to provide insights into how an application is performing through a combination of compiler infrastructure, a runtime library, and post-tracing analysis tools.

In more detail, XRay consists of the following:

- Changes to the compiler to insert small "no-op" code sequences in function entry and exit points, which at runtime get patched to enable function-level tracing.
- A runtime library for logging function entry/exit events that dynamically prunes recorded events to maximize the 'value-per-MB' of the recorded traces. .
- A tool that reconstructs timestamped function call trees from the XRay traces.
- Analysis tools that take the function call logs and highlight potential issues such as lock contention.

This allows engineers to build and deploy a single XRay-instrumented binary to production that can be used both for standard deployment (tracing turned off, overhead minimal) as well as for debugging . When debugging live systems, the runtime library gathers trace data on demand which provides very detailed function call traces, as well as input to analysis tools.

XRay is one part of a suite of tools for debugging systems at Google. It has been used successfully to identify latency issues in storage systems, web serving, ads serving, as well as other services in production.

How XRay Works

XRay relies on compiler changes to insert no-op sleds[2] in function entry/exits, and record those locations in tables encoded in the object files. At runtime, if XRay is disabled, these no-op sleds are executed as-is and add minimal execution overhead. However, if XRay is enabled, the XRay runtime library overwrites these no-ops with calls to instrumentation code that logs function entry/exit information to in-memory buffers. These logs also contain

[2] These "no-op sleds" are a few bytes of code emitted by the compiler that do nothing.

cycle-counter timestamps and enough metadata to reconstruct the program's operation in post-processing.

XRay works in fully multithreaded programs. Turning tracing on introduces code at runtime to insert instrumentation at the right sections of the program code. Turning tracing off undoes these changes. XRay keeps the program semantics intact without having to force single processor mode or stopping execution of the program while the instrumentation is being added and enabled.

XRay Overheads

To minimize the costs of the no-op sleds inserted at function entry and exit points, XRay employs heuristics to determine which functions to instrument. XRay will only instrument functions that are either:

- At least N instructions. By default N is 200, based on an assumption that really small functions (without loops in them) are unlikely to take a significant amount of time and therefore are not worth instrumenting. This can be tuned by flags at compilation time to tweak the number of instructions to use as a threshold.
- Containing non-trivial loops. This assumes that loops may be introducing a variable amount of time, and that users in general would like to see this variability accounted for by XRay.

When tracing is enabled, we've measured an increase of between 20-40% in CPU usage with a proportional in execution time. Because of this known overhead, we usually enable XRay tracing and collection for brief periods of time to minimise the effect on live running systems and collect useful data from services under load. Typically we collect around a few seconds of data, which requires around 200MB of memory on a busy server.

We also typically only see up to a 2% increase in binary size with the XRay instrumentation points.

Implementation Details

XRay has a few moving parts that work together to provide an accurate picture of what functions are running in a server. We cover each of those moving parts in detail in the following sections.

Compiler-inserted Instrumentation Points

At Google, we have patches on top of GCC that implement the XRay instrumentation point insertion heuristics. The result of these patches yield code that in x86 assembler looks like:

```
local_block_sled_0:
  jmp . + 0x09
  (9 bytes worth of nops)
... # function prologue starts, followed by the body.
... # function epilogue starts, just before ret...
local_block_sled_1:
  retq
  (10 bytes worth of nops)
```

Each object file then has two named sections (`__function_patch_prologue`, `__function_patch_epilogue`) where the various addresses of the sled labels (`local_block_sled_`*M*) are kept. These sections gets consolidated when the binary is linked statically.

As mentioned earlier, we apply a heuristic on each function encountered by the compiler when emitting the assembler. The heuristic can be thought of as an implementation of the following pseudocode:

```
if instruction_count(function) > threshold
   OR has_non_trivial_loop(function):
  xray_instrument(function)
emit_function(function)
```

The threshold is user-controllable as a command-line option.

XRay also supports attributes that can be added to functions to ensure that they are always instrumented if the XRay option is enabled in the compiler. These attributes take the form:

```
__attribute__((always_patch_for_instrumentation))
void Function() {
  ...
}
```

These attributes can be explicitly provided for functions that should always be instrumented (e.g. functions for locking mutexes for lock contention analysis) and treated as special by the runtime library.

Runtime Logging and Support Library

At runtime, XRay provides APIs to install the appropriate patch instructions into the instrumentation sleds. This API also includes a mechanism for installing a logging function called for every function entry and exit sled. The library loads the instrumentation map and goes through and patches the following instructions:

```
#(1) for function entry sleds
mov $<function id>, %r10d
call __xray_FunctionEntryStub

#(2) for function exit sleds
mov $<function id>, %r10d
jmp __xray_FunctionExitStub
```

The process of patching and un-patching the instrumentation sleds is:

1. We go through each entry in the instrumentation map either loaded at runtime from an external file or from the special sections in the binary and do the following:
    a. For the function entry sleds, we replace the jmp and 9-byte nops with `#(1)` above by writing the last 9 bytes first (the `call`), then atomically writing the first two bytes (the `mov`) over the `` `jmp . + 0x09` `` instruction.
    b. For the function exit sleds, the process is similar to preserve correct execution in multithreaded environments.
2. Once all the sleds have been patched, we atomically set a flag at runtime checked by both the `__xray_FunctionEntryStub` and `__xray_FunctionExitStub` functions to enable function call logging.
3. When tracing is explicitly turned off, we do either one of the following:
    a. If asked to explicitly "unpatch" the code, we go through the instrumentation map and reverse the process done in 2, 1.a, and 1.b.
    b. If we only disable the logging but keep the instrumentation in place, we atomically set the flag checked by both `__xray_FunctionEntryStub` and `__xray_FunctionExitStub` to false. As an optimisation we also change the first instruction in both `__xray_FunctionEntryStub` and `__xray_FunctionExitStub` to be an immediate `` `ret` `` to further lower the cost of the instrumentation functions.

This process has been tuned to be safe on multithreaded applications, allowing the process to continue making progress while instrumentation is being added and removed. The atomic updates above makes sure that the program counter for all cores always points to a valid instruction.

The choice in 3 above is determined by whether the user just wants to disable the logging functionality for a period of time. This has been useful for periodic tracing of the same application running in production at different times. Collecting sample traces is one way of gathering enough data for offline analysis.

The `__xray_FunctionEntryStub` and `__xray_FunctionExitStub` functions call logging functions for entry and exit logging respectively. These logging functions do the following:

`LogFunctionEntry`:

- Get a cycle counter (`RDTSC` in x86)
- Log an entry to a thread-specific buffer of the following form:
  - Function ID, Function Entry Identifier, Cycle counter delta
- Returns to the calling function.

`LogFunctionExit`:

- Get a cycle counter (`RDTSC` in x86)
- Logs an entry of the form:
  - Function ID, Function Exit Identifier, Cycle counter delta
- Returns to the calling function.

Each thread will be provided an 8 Kilobyte (KB) buffer the first time a thread needs to write a log entry. Each 8KB block is dedicated to a thread, eliminating the need to synchronise among multiple threads and implicitly maps blocks to threads without having to record the thread ID for every record. These blocks have a 192-byte header and 1000 8-byte (64-bit) chunks for each record. Each entry as described above will be:

<metadata (8 bits), function id (24 bits), 32 bits Cycle counter delta>

The 32 bits come from a 64 bit TSC discarding the 10 least significant and highest 22 bits. This represents microsecond accuracy for each log entry in this buffer. The 8 bits of metadata indicate whether this entry is a function entry, function exit, or a metadata entry. For example, a metadata entry is used to indicate when a TSC wraps around. Further optimisations also allow for writing just 4 bytes for a record (6 bits metadata, 19 bits function id, and 8 bits cycle counter delta).

The header bytes include information about the thread (the processid and thread id in Linux), and a reference 64-bit TSC and a wallclock entry in microsecond precision (from gettimeofday). These are used in post-processing to stitch together the timestamps from TSC in microsecond granularity to allow conversion to human-readable timestamps relative to some reference wallclock time.

The logging functions by default prune records that are less than 5 microseconds equivalent in walltime deduced from the cycle counter deltas. This allows XRay to retain only records that have a measurable impact in walltime. This behaviour is based on internal experimentation and is configurable.

The remaining part of the logging implementation is a "dump data" function, which gathers all the thread-specific buffers used by the tracing system and either makes the data available in memory as an iterable set of 8K chunks of data or writes it out to different files with a predefined naming convention. The XRay library uses a filename provided via a commandline option. Each thread's chunks will be written out into different files containing the thread id. The files will contain the raw XRay trace logs which will later be analysed by a separate set of tools.

When the thread runs out of space for a given block it returns this block to, and gets another block from, a circular buffer of blocks.

Conversion and Analysis Tools

XRay provides a library to allow for reconstructing function call trees from the raw trace data into a canonical data format for events. We have internal tools that convert the event data files into different formats:

- HTML Trace File. We turn the data into a standalone interactive spreadsheet-like HTML file that shows all the function calls that were recorded, which threads they showed up in, in which RPCs they were being processed in, and how long each function call took. We've implemented some optimisations in terms of encoding the data so that search and filtering functionality is efficiently implemented in self-hosted JavaScript. This has been used to debug both rare and common performance issues (expensive copies of large objects, memory allocation/deallocation patterns, etc.)
- Domain specific analysis results. We also have internal tooling that allows for writing analysers using an analysis framework to perform domain specific pattern matching. We've used this to find certain call patterns that usually show common problems (slow writes, lock contention, thread residency analysis, etc).

In Google, XRay is used as one data source within a more comprehensive performance analysis and debugging suite of tools.

Current Work and Future Plans

We are committed to making XRay available as open source software, and as such we are engaging the LLVM[3] community to get all of the pieces of XRay released -- changes to the compilers, the runtime library, and ways to work with the generated traces. We are also committing to engaging open source communities to build the tools to make XRay data more useful. We believe that function call tracing is one tool in a toolbox of debugging and performance analysis tools that every developer should have available.

Once XRay is released and widely available, we are going to work with developers willing to port XRay into other machine architectures and operating systems. We will actively be involved in maintaining XRay and improving it based on feedback from the open source community. Because XRay is used at Google to find hard-to-debug problems, we will make sure that XRay continues to be an actively maintained project for the long term.

We are also going to engage potential users and tool builders that want to make performance debugging a more pleasant experience for the C/C++ developer community as well as other languages.

Acknowledgements

XRay was done at Google by Harshit Chopra, Robert Bowdidge, Sanjay Bhansali, and Vlad Losev with additional contributions by David Goldblatt . Without their efforts and investment in building XRay, the team currently working with it (Alistair Veitch, Dean Berris, Eric Anderson, Nevin Heintze, Ning Wang) and other Google teams would have a very hard time finding performance bugs in production systems. We also would like to thank Eric Christopher, Chandler Carruth, Matt Austern, and Andrew Fikes who reviewed an early draft of this white paper.

[3] The LLVM Project is at http://llvm.org/.